Implicit Polymorphic Type System for the Blue Calculus

Silvano Dal Zilio

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implicit Polymorphic Type System for the Blue Calculus

Silvano Dal-Zilio

N° 3244

Septembre 1997

THÈME 1

ISSN 0249-6399

Implicit Polymorphic Type System for the Blue Calculus

Silvano Dal-Zilio*

Thème 1 — Réseaux et systèmes
Projet MEIJE

Rapport de recherche n° 3244 — Septembre 1997 — 31 pages

Abstract: The Blue Calculus is a direct extension of both the lambda and the pi calculi. In a preliminary work from Gérard Boudol, a simple type system was given that incorporates Curry's type inference for the lambda-calculus. In the present paper we study an implicit polymorphic type system, adapted from the ML typing discipline. Our typing system enjoys subject reduction and principal type properties and we give results on the complexity for the type inference problem. These are interesting results for the blue calculus as a programming notation for higher-order concurrency.

Key-words: pi-calculus, blue-calculus, type, polymorphism

(Résumé : tsvp)

This work was partially supported by France Télécom, CTI-CNET 95-1B-182 Modélisationn des systèmes Mobiles.

* Email : Silvano.Dal_Zilio@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – Télécopie : 04 93 65 77 65

Système de type polymorphe pour le Calcul Bleu

Résumé : Le Calcul Bleu est une extension directe à la fois du pi-calcul et du lambda-calcul. Dans un travail précédent, Gérard Boudol à donné un système de type simple pour ce calcul qui comprend le système de type simple de Curry pour le lambda-calcul. Dans ce rapport, nous étudions un système de type implicite et polymorphe qui s'inspire du typage du langage ML. Nous montrons que ce système de type possède les propriétés de "subject reduction" et de type principal, ce qui conforte notre opinion que le calcul bleu est un modèle bien adapté à la définition d'un langage de programmation concurrent d'ordre supérieur.

Mots-clé : pi-calcul, calcul bleu, type, polymorphisme

1 Introduction

The purpose of this paper is to define a polymorphic type discipline for the blue calculus ($\pi^\star$), a direct extension of both the λ-calculus and the π-calculus. As in Curry's work, we start from an untyped calculus, i.e., without type annotations, and we define a type system on top of it. Types are used as a means of identifying a class of terms whose evaluation is guaranteed to be safe, but Curry's type system is limited in the sense that some "reasonable" terms are ill-typed. The Hindley-Milner type system, used in the definition of many higher-order functional languages like HASKELL or ML, partially overcomes this limitation by introducing parameterization on types. In [8], a simple type system was given that incorporates Curry's type inference for the λ-calculus (see table 1 on page 9). Therefore, it is natural to encounter the same limitation. Our goal is to apply to the blue calculus the same program that benefited ML. In this paper we present a type system that combines both safety: strong and static type checking, and flexibility: parametric polymorphism, implicit type assignment and decidable type inference.

To get an idea of the blue calculus, let us see how the weak call-by-name λ-calculus can be encoded in the asynchronous π-calculus [4, 7]. We suppose the set of λ-calculus variables is a subset of the π-calculus set of channel names. Lambda terms are given by the usual grammar

$$M ::= x \mid \lambda x.M \mid (M\ M)$$

and the lazy reduction relation $M \to_l M'$ is given by the two rules

$$\begin{array}{l}(\lambda x.M)\ N \to_l M[N/x] \\ M \to_l M' \Rightarrow (M\ N) \to_l (M'\ N)\end{array}$$

The translation, $[\![M]\!]_u$, from λ to π, takes as extra argument a channel name, u, representing the "location" at which the λ-term will have access to its next argument

$$\left\{\begin{array}{lcl}[\![x]\!]_u & =_{\mathrm{def}} & \bar{x}u \\ [\![\lambda x.M]\!]_u & =_{\mathrm{def}} & u(x,v).[\![M]\!]_v \\ [\![(M\ N)]\!]_u & =_{\mathrm{def}} & (\nu v)([\![M]\!]_v \mid (\nu x)(\bar{v}xu \mid !x(w').[\![N]\!]_{w'})) \\ & & \\ [\![(\lambda x.M)\ N]\!]_u & = & (\nu v)(v(x,w).[\![M]\!]_w \mid (\nu x)(\bar{v}xu \mid !x(w').[\![N]\!]_{w'})) \\ & \to & (\nu vx)([\![M]\!]_u \mid !x(w').[\![N]\!]_{w'})\end{array}\right.$$

The process $D =_{\mathrm{def}} x(w').[\![N]\!]_{w'}$ can be interpreted as an agent, located at x, waiting for a location u to trigger the process $[\![N]\!]_u$. For example, in the reduction

$$[\![x]\!]_u \mid x(w').[\![N]\!]_{w'} \to_\pi [\![N]\!]_u$$

the output $\bar{x}u$ triggers a process "equivalent" to N waiting for its next argument in u. We choose to denote this special agent

$$D =_{\mathrm{def}} (x \Leftarrow (w')[\![N]\!]_{w'})$$

and we call it a *declaration* in the following. We also consider the associated reduction rule

$$\bar{x}u \mid (x \Leftarrow (y)P) \rightarrow P[u/y]$$

which can be further decomposed in the more elementary reduction steps fetching and "small" β-reduction[1]

$$\bar{x}u \mid (x \Leftarrow (y)P) \rightarrow_\rho ((y)P\ u) \quad \text{and} \quad ((y)P\ u) \rightarrow_\beta P[u/y]$$

Therefore, the translation of redex $((\lambda x.M)\ N)$ reduces to a process equivalent to $[\![M]\!]_u$ in parallel with an agent perpetually offering N at x. This situation is comparable to the call-by-need calculus of Ariola *et al* [1], a λ-calculus extended with the "let" operator together with the reduction rule

$$(\lambda x.M)\ N \rightarrow \text{let}\, x = N \,\text{in}\, M \tag{1}$$

A comparison with β-reduction $(\lambda x.M)\ N \rightarrow M[N/x]$ would have been less satisfactory since, in our translation, only an "access" to N is transmitted and not the term itself.

The next step is to identify $\bar{x}u$ with application and $(x)N$ with small λ-abstraction (a comparison already done by Milner in [28]). We have by definition that $[\![x]\!]_u =_{\text{def}} \bar{x}u$. Abstracting the extra argument u, it follows that $(u)([\![x]\!]_u) = (u)(\bar{x}u)$, which is η-equivalent to $\bar{x}$. Likewise, if we denote $\{\![M]\!\} =_{\text{def}} (u)([\![M]\!]_u)$, we have that $(w')([\![N]\!]_{w'}) =_{\text{def}} (w')(\{\![N]\!\}\ w') \approx_\eta \{\![N]\!\}$. Introducing a new notation for the infinitely available resource N at x, namely $(x = N) =_{\text{def}} !(x \Leftarrow N)$, it follows that the translation of redex $(\lambda x.M)\ N$ becomes

$$[\![(\lambda x.M)\ N]\!]_u =_{\text{def}} (\nu v)((v \Leftarrow (\lambda x)\{\![M]\!\}) \mid (\nu x)(\bar{v}x \mid (x = \{\![N]\!\})))$$

Reduction is again the combination of the two previous basic reduction steps, communication (fetching) and small β-reduction

$$\begin{array}{rll} [\![(\lambda x.M)\ N]\!]_u & =_{\text{def}} & (\nu v)((v \Leftarrow (\lambda x)\{\![M]\!\}) \\ & & \quad \mid (\nu x)(\bar{v}x \mid (x = \{\![N]\!\}))) \\ & \rightarrow_\rho & (\nu vx)(((\lambda x)\{\![M]\!\}\ x) \mid (x = \{\![N]\!\})) \\ & \rightarrow_\beta & (\nu vx)(\{\![M]\!\} \mid (x = \{\![N]\!\})) \end{array}$$

Together with Equation (1), this gives a definition of the let-construct in the blue calculus

$$\text{let}\, x = N \,\text{in}\, M \quad =_{\text{def}} \quad (\nu x)(M \mid (x = N)) \tag{2}$$

The blue calculus syntax and operational semantic is very close to what has been introduced in this translation from weak call-by-name λ-calculus to π. We find both λ and

[1] small is used to distinguish Church's β rule, such that a term replace a variable, from the weaker relation $\rightarrow_\beta$, such that variable are only instantiated with another variable.

π operators: application and abstraction in the applicative part and parallel ($|$) and new names declaration (ν) in the concurrent one. We also find new constructors for resources declaration: $\langle u \Leftarrow P \rangle$ and $\langle u = P \rangle$, together with the reduction rule

$$u\ z_1 \dots z_n \mid \langle u \Leftarrow P \rangle \rightarrow_\rho P\ z_1 \dots z_n$$

Equation (2) is another contribution of this translation. It links declarations and let bindings, the construct at the heart of polymorphism in the Hindley-Milner type system. This relation is extended to types in Section 3.1 and is the main theme that permits us to understand the different type system proposed in this paper.

The let-construct is definable in our calculus, we do not have to consider a new operator. There is a sensible differences with the λ-calculus though. One should remark that the let operator is usually associated with a call-by-value evaluation mechanism whereas $\pi^\star$ is associated with call-by-name[2]. Therefore, we have to differentiate the two mechanisms bounded in the ML let-construct

Value sharing $\mathrm{let}\, x = N \,\mathrm{in}\, M \rightarrow M[N/x]$ iff N is a value whereas, in $\pi^\star$, computation of N is duplicated in $(\nu x)(M \mid (x = N))$ if x appears more than once in M.

Polymorphism Type generalization is restricted to variable bounded in a let-construct.

It is interesting to compare this remark with the distinction made by Leroy in [24, 25] between a construct, "let val", for value sharing and a construct, "let name", for type generalization. The same distinction is also introduced in [17], where it is stated that, in this variant of ML, it becomes possible to define a correct "variant call-by-value CPS transform" for a polymorphic type assignment system.

In Section 2 we briefly present $\pi^\star$ and its operational semantics. And in Section 3 we present the polymorphic type system and the problem of polymorphic recursion. We define two type systems adapted from type discipline for ML: one ($\vdash_{\mathtt{D}}$) inspired by the Damas-Milner type system and the other (denoted $\vdash_{\mathtt{M}}$) inspired by Milner-Mycroft's one. In Section 4 and 5 we prove subject reduction and principal type property for $\vdash_{\mathtt{M}}$, the most expressive of these systems. In Section 6, we define a new constructor that binds the scope of declarations and give a new type system, $\vdash_{\forall\mathtt{B}}$, for the extended calculus. In Section 7 we prove that $\vdash_{\mathtt{D}}$ and $\vdash_{\mathtt{M}}$ both give an undecidable type assignment problem whereas typability in $\vdash_{\forall\mathtt{B}}$ is decidable.

2 The semantics of the blue calculus

We assume two denumerable sets of names, $\mathcal{V}$ for variables, ranged over by $x, y, \dots$ and $\mathcal{R}$ for references, ranged over by $u, v, \dots$. We distinguish four syntactic categories in the grammar of our calculus as

[2] there is no rule that allow reduction under a declaration

$$\begin{array}{rcll} a & ::= & \mathit{constants} \mid x \mid u & \text{values} \\ D & ::= & \langle u \Leftarrow P \rangle \mid \langle u = P \rangle & \text{declarations} \\ A & ::= & a \mid (\lambda x)P \mid (P\ a) & \text{agents} \\ P & ::= & A \mid D \mid (P \mid P) \mid (\nu u)P & \text{processes} \end{array}$$

We denote $\mathrm{fn}(P)$ to be the free names of P. A bound name is either a (λ) abstracted variable or a restricted reference. Reference u is restricted in $(\nu u)P$ but free in declarations $\langle u \Leftarrow P \rangle$ or $\langle u = P \rangle$, $\mathrm{decl}(P)$ denotes the free references in the subject of a declaration. In particular $\mathrm{decl}(\langle u \Leftarrow R \rangle) = \{u\} \cup \mathrm{decl}(R)$.

The operational semantics is presented using the chemical metaphor [2]. In particular, structural equivalence over processes, denoted $\rightleftharpoons$, is defined as the combination of two relations, cooling $\rightharpoonup$ and heating $\leftharpoondown$ (see Table 1). Axioms for the heating relations can be deduced from those of $\rightleftharpoons$ reading the rules from left to right (respectively right to left for cooling). The rules (folding) and (replication) can conveniently be added, but are not necessary.

Apart from for declarations, the constructors operational behaviour (given Table 1) should be clear. $\langle u \Leftarrow P \rangle$ can be thought as a resource accessible through a call to reference u. Rule (ρ), in the operational semantic, implements the "resource fetching" mechanism

$$(u\ z_1 \dots z_n \mid \langle u \Leftarrow P \rangle) \rightarrow P\ z_1 \dots z_n$$

Likewise, $\langle u = P \rangle$ may be interpreted as an "inexhaustible resource", related to $\langle u \Leftarrow P \rangle$ in the same way that $!u(x).P$, the π-calculus replicated input, is related to $u(x).P$. To understand the connections between resources, the λ and the π-calculus it is interesting to study how declarations are reminiscent of the λ-calculus with resources and its deterministic version, the λ-calculus with multiplicities [5, 6, 7, 23], defined by Gérard Boudol in connection with the encoding of the λ-calculus in π.

It should be noted that, contrary to the original presentation made in [8], references are not first class in our calculus[3]. Although they can appear under a λ-abstraction, they can't be abstracted. For example, $(\lambda u)\langle u \Leftarrow P \rangle$ is not a valid process. This restriction implies that no new declarations on a given reference can be dynamically created. We does not lose expressive power though. Indeed, from a π-calculus viewpoint, this restriction is equivalent to the limitation that a name received during an input cannot be used as an input channel. This restriction is present in Michele Boreale's π^i_a-calculus [3], and it has been shown that the asynchronous π-calculus of Boudol [4, 19] can be faithfully encoded in its fragment π^i_a.

To conclude, let us see, in some examples, how the blue calculus takes the best from both the functional and the process calculi world. It is possible to give example familiar

[3] in fact, the same discussion appears in conclusion of this paper

Table 1 structural equivalence and reduction

structural equivalence:

$P \rightleftharpoons Q$	$(P \equiv_\alpha Q)$
$P \mid Q \rightleftharpoons Q \mid P$	
$(P \mid Q) \mid R \rightleftharpoons P \mid (Q \mid R)$	
$(P \mid Q)\ v \rightleftharpoons (P\ v) \mid (Q\ v)$	
$(\nu v)P \mid Q \rightleftharpoons (\nu v)(P \mid Q)$	$(v \notin \mathrm{fn}(Q))$
$(\nu u)P\ v \rightleftharpoons (\nu u)(P\ v)$	$(u \neq v)$
$\langle u = P\rangle\ a \rightleftharpoons \langle u = P\rangle$	
$\langle u \Leftarrow P\rangle\ a \rightleftharpoons \langle u \Leftarrow P\rangle$	
$\langle u = P\rangle \rightleftharpoons \langle u \Leftarrow (P \mid \langle u = P\rangle)\rangle$	folding/unfolding
$\langle u = P\rangle \rightleftharpoons \langle u \Leftarrow P\rangle \mid \langle u = P\rangle$	replication

reduction:

$(\lambda x)P\ v \rightarrow P[v/x]$	(β)
$(u \mid \langle u \Leftarrow P\rangle) \rightarrow P$	(ρ)
$(u \mid \langle u = P\rangle) \rightarrow P \mid \langle u = P\rangle$	$(!\rho)$
$P \rightarrow P' \Rightarrow (P\ u) \rightarrow (P'\ u)$	context rules
$P \rightarrow P' \Rightarrow (P \mid Q) \rightarrow (P' \mid Q)$	
$P \rightarrow P' \Rightarrow (\nu u)P \rightarrow (\nu u)P'$	
$P \rightarrow P'$ and $P \rightleftharpoons Q \Rightarrow Q \rightarrow P'$	

to people from the λ-calculus world. Booleans encodings for example: $T =_{\text{def}} (\lambda xy)x$ and $K =_{\text{def}} (\lambda xy)y$, or pairs, $[P, Q]$, such that

$$[P, Q] =_{\text{def}} (\nu pq)((\lambda b)(b\ p\ q) \mid \langle p \Leftarrow P\rangle \mid \langle q \Leftarrow Q\rangle)$$

$$[P, Q]\ \ T \to P \text{ and } [P, Q]\ \ K \to Q$$

We can also define classical concurrent processes, like process-based lists, non-deterministic choice ($\oplus$) or single state buffer (*buff*)

$$\oplus =_{\text{def}} (\lambda xy)((\nu u)(u \mid \langle u \Leftarrow x\rangle \mid \langle u \Leftarrow y\rangle))$$

$$\mathit{buff} =_{\text{def}} (\nu b)(b \mid \langle b = \langle \mathit{put} \Leftarrow (\lambda x)\langle \mathit{get} \Leftarrow (x \mid b)\rangle\rangle\rangle)$$

Finally, some $\pi^\star$ specific operators are also definable. Like "higher-order" application or recursion

$$(P\ Q) =_{\text{def}} (\nu u)(P\ u \mid \langle u = Q\rangle) \quad (u \text{ fresh})$$

$$\text{rec}\, x.P\ =_{\text{def}}\ (\nu x)(\langle x = P\rangle \mid x)$$

3 The type system

In the polyadic π-calculus [28], contrary to the monadic case where simple channel names are exchanged during communication, channels carry tuples. This led to a new behaviour: tuples sent along a channel may not have the same length as the tuple expected by the receiver. A situation comparable to runtime failure. To prevent this situation, Milner defined a channel-use discipline, called *sorts*, which is the foundation of many type systems for the π-calculus [32, 37] and for programming languages developed on its model (PICT [35], the join-calculus [16], etc).

Since communication in $\pi^\star$ is a combination of resource fetching and β-reduction steps there is no notion of a polyadic calculus. There is also no corresponding operational notion of runtime failure. But the notion of types for our calculus arises naturally from the λ-calculus, especially if one wants to introduce concrete values. Moreover, Boudol proved that his Curry-Hindley style type system for $\pi^\star$ (Table 1) reconciles both the λ-types and the π-sorts paradigms.

We assume the reader is familiar with notations for the polymorphic type discipline of ML. We use β to denote type variables (also called non-generic or applicative variables) and α to denote generic type variables. By convention we use τ and ω to range over simple (monomorphic) types and σ, ρ over type schemes (polymorphic types). $\text{fn}(\tau)$ is the set of free types variables and $\Gamma_{|x}$ denotes a type environment equal to Γ everywhere save for x, where it is undefined.

Type System 1 Curry-style type system for $\pi^\star$ ($\vdash_{\mathtt{B}}$)

$$\frac{\Gamma(u) = \tau}{\Gamma \vdash_{\mathtt{B}} u : \tau}\ (\text{taut})$$

$$\frac{\Gamma \vdash_{\mathtt{B}} P : \tau \quad \Gamma(x) = \tau'}{\Gamma_{|x} \vdash_{\mathtt{B}} (\lambda x)P : \tau' \to \tau}\ (\text{abs}) \qquad \frac{\Gamma \vdash_{\mathtt{B}} P : \tau' \to \tau \quad \Gamma(u) = \tau'}{\Gamma \vdash_{\mathtt{B}} (P\ u) : \tau}\ (\text{app})$$

$$\frac{\Gamma \vdash_{\mathtt{B}} P : \tau \quad \Gamma \vdash_{\mathtt{B}} Q : \tau}{\Gamma \vdash_{\mathtt{B}} P \mid Q : \tau}\ (\text{par}) \qquad \frac{\Gamma \vdash_{\mathtt{B}} P : \tau}{\Gamma_{|x} \vdash_{\mathtt{B}} (\nu x)P : \tau}\ (\text{new})$$

$$\frac{\Gamma \vdash_{\mathtt{B}} P : \tau \quad \Gamma(u) = \tau}{\Gamma \vdash_{\mathtt{B}} \langle u \Leftarrow P \rangle : \omega}\ (\text{decl}) \qquad \frac{\Gamma \vdash_{\mathtt{B}} P : \tau \quad \Gamma(u) = \tau}{\Gamma \vdash_{\mathtt{B}} \langle u = P \rangle : \omega}\ (!\text{decl})$$

$$\begin{array}{lcll} v & ::= & \textit{basetypes} \mid \alpha \mid \beta & \\ \tau & ::= & v \mid (\tau \to \tau) & \text{simple types} \\ \sigma & ::= & \tau \mid \forall\alpha.\sigma & \text{type schemes} \end{array}$$

Definition 3.1 (Generalization of a type and subsumption)
Given a set V of type variables, generalization of a simple type τ is the type scheme $\mathrm{Gen}_V(\tau) =_{\mathrm{def}} \forall\tilde{\alpha}.\tau$, *where $\tilde{\alpha}$ are the free generic type variables in τ not belonging to V. We use* $\mathrm{Gen}_\Gamma(.)$ *as a shorthand for* $\mathrm{Gen}_{\mathrm{fn}(\Gamma)}(.)$. *The subsumption relation between type schemes and types, $\prec$, is such that $\sigma \prec \tau$ iff there exist some simple types τ' and $\tilde{\omega}$, and variables $\tilde{\alpha}$ such that $\sigma = \forall\tilde{\alpha}.\tau'$ and $\tau = \tau'[\tilde{\omega}/\tilde{\alpha}]$.*

We present a syntax-directed type inference system (see table 2 on the next page) in the same vein as [12]. Note that, contrary to [14], we give simple types to processes and not type schemes. An enjoyable propriety of such system is that the structure of a valid typing sequent $\Gamma \vdash P : \tau$ is P.

For convenience, we neglect to give a typing rule for multiple/replicated declarations ($\langle u = P \rangle$) since, like in $\vdash_{\mathtt{B}}$, it is essentially the same as for simple/linear declarations ($\langle u \Leftarrow P \rangle$). Moreover, in a calculus with structural laws (replication) and (folding) (table 7), one can derive the former rule from the latter. The rules for declarations deserve further comment. A sequent $\Gamma, u : \sigma \vdash P : \tau$ means that every occurrence of u in P has (possibly different) type ω such that $\sigma \prec \omega$. Suppose that the process Q is a resource accessible from u in P, e.g., $P = \mathcal{C}[u] \mid \langle u \Leftarrow Q \rangle$. Therefore $\langle u \Leftarrow Q \rangle$ should have type τ and Q should have type τ_q such that $\tau \prec \tau_q$. Therefore, we must be able to assigned any type to a declaration. This is the meaning of type ω in rule (decl+).

Type System 2 Polymorphic type system for $\pi^\star$ ($\vdash$)

$$\frac{\Gamma(x) \prec \tau}{\Gamma \vdash x : \tau}\ (\text{taut})$$

$$\frac{\Gamma \vdash P : \tau \quad \Gamma(x) = \tau'}{\Gamma_{|x} \vdash (\lambda x)P : \tau' \to \tau}\ (\text{abs}) \qquad \frac{\Gamma \vdash P : \tau' \to \tau \quad \Gamma(u) \prec \tau'}{\Gamma \vdash (P\ u) : \tau}\ (\text{app})$$

$$\frac{\Gamma \vdash P : \tau \quad \Gamma \vdash Q : \tau}{\Gamma \vdash P \mid Q : \tau}\ (\text{par}) \qquad \frac{\Gamma \vdash P : \tau}{\Gamma_{|x} \vdash (\nu x)P : \tau}\ (\text{new})$$

$$\frac{\Gamma \vdash P : \tau \quad \Gamma(u) = \mathrm{Gen}_\Gamma(\tau)}{\Gamma \vdash \langle u \Leftarrow P \rangle : \omega}\ (\text{decl}+)$$

We implicitly assume here that we are dealing with terms up to α-conversion. Otherwise we could not be able to type an example like $(\lambda x)(\lambda x)x$ for instance. Thus we, can assume that our type inference system is enriched with a typing rule that allows two α-equivalent processes to share the same types and, for the rest of this paper, we assume that bound names never clash.

Type System 3 α conversion

$$\frac{\Gamma \vdash P : \tau \quad P \equiv_\alpha Q}{\Gamma \vdash Q : \tau}\ (\text{alpha})$$

3.1 Polymorphic recursion

For our analysis, only a core fragment of ML is needed. It's a λ-calculus augmented with the let-construct and the usual fix-point operator, namely $\mathrm{rec}\, x.M$. We recall that we distinguish $(\lambda x)P$, the blue calculus "small" abstraction, and $\lambda x.M$, the stronger λ-calculus abstraction, that allows substitution of terms over variables. We refer the reader to the note in page 4 and to [28, 8] where the same discussion appears. Terms of core-ML are defined by the grammar

$$M ::= x \mid (M\,M) \mid \lambda x.M \mid \mathrm{let}\, x = N \,\mathrm{in}\, M \mid \mathrm{rec}\, x.M$$

A type inference system, based on the one given by Damas in his thesis [14], is presented in Table 4. A characteristic of the Damas-Milner Calculus is that occurrences of a function inside the body of its (recursive) definition can only be used monomorphically. To overtake this limitation, Mycroft [29] suggested a typing rule for polymorphic recursive definitions

(see Table 5 and rule (letrec +), table 8 on page 20). We differentiate those two type systems by denoting with $\vdash_{\mathtt{D}}$ sequents in the presence of (rec) rule and $\vdash_{\mathtt{M}}$ whenever rule (rec +) is used.

Type System 4 Type system for core-ML without recursion

$$\frac{\Gamma(x) = \sigma \qquad \sigma \prec \tau}{\Gamma \vdash x : \tau}\ (\text{taut}) \qquad \frac{\Gamma \vdash M' : \tau' \qquad \Gamma_{|x}, x : \mathrm{Gen}_{\Gamma}(\tau') \vdash M : \tau}{\Gamma \vdash \mathrm{let}\, x = M' \,\mathrm{in}\, M : \tau}\ (\text{let})$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma(x) = \tau'}{\Gamma_{|x} \vdash \lambda x.M : \tau' \to \tau}\ (\text{abs}) \qquad \frac{\Gamma \vdash M : \tau' \to \tau \qquad \Gamma \vdash M' : \tau'}{\Gamma \vdash (M\ M') : \tau}\ (\text{app})$$

Type System 5 Two typing rules for recursive definitions in ML

Damas-Milner type system ($\vdash_{\mathtt{D}}$) $\qquad \dfrac{x : \tau, \Gamma_{|x} \vdash_{\mathtt{D}} M : \tau}{\Gamma_{|x} \vdash_{\mathtt{D}} \mathrm{rec}\, x.M : \tau}\ (\text{rec})$

Milner-Mycroft type system ($\vdash_{\mathtt{M}}$) $\qquad \dfrac{x : \mathrm{Gen}_{\Gamma_{|x}}(\tau), \Gamma_{|x} \vdash_{\mathtt{M}} M : \tau}{\Gamma_{|x} \vdash_{\mathtt{M}} \mathrm{rec}\, x.M : \tau}\ (\text{rec}+)$

The rule in Damas-Milner (DM) type system is conceivable if one considers recursive definition as the result of applying a fixpoint operator to an abstraction, i.e., $\mathrm{rec}\, x.M \simeq (Y\ \lambda x.M)$, where Y's type is $\forall\alpha.(\alpha \to \alpha) \to \alpha$. On the other hand, in the Milner-Mycroft (MM) type inference system, the definition of a function f can contain recursive calls to f at different types. This ability is called *polymorphic recursion*. It was shown in Section 2 that the rec-operator is derivable in the blue calculus

$$\mathrm{rec}\, x.P \ =_{\mathrm{def}}\ (\nu x)(\langle x = P\rangle \mid x)$$

this gives derived rules for recursion in our polymorphic type system (see table 6 on the next page). Looking at the corresponding rules in ML, it appears that our type inference system is closer to Mycroft spirit.

The Milner-Mycroft type inference system is proved sound and it preserves the principal type property of DM calculus. But, contrary to Damas type system, type inference is not a decidable problem. Automatic and practical type inference for ML has been achieved for the earlier implementation [27, 13] and was a major advantage of ML with respect to other applicative programming languages. However, results concerning the type reconstruction complexity are more recent. For almost ten years programmers thought this problem had polynomial time complexity until Maison [26] proved in 1990 that the polymorphic type inference problem is complete for exponential time. Likewise, the MM type system from 1984,

Type System 6 Damas and Mycroft style typing rule for declaration and associated derived rules for recursive definitions

$$\frac{\Gamma_{|u}, u : \tau \vdash_{\mathtt{D}} P : \tau}{\Gamma_{|u}, u : \mathrm{Gen}_{\Gamma_{|u}}(\tau) \vdash_{\mathtt{D}} \langle u \Leftarrow P \rangle : \omega} \text{ (decl)}$$

$$\frac{\Gamma_{|u}, u : \tau \vdash_{\mathtt{D}} P : \tau \qquad \mathrm{Gen}_{\Gamma_{|u}}(\tau) \prec \tau'}{\Gamma_{|u} \vdash_{\mathtt{D}} \mathrm{rec}\, u.P : \tau'} \text{ (rec)}$$

$$\frac{\Gamma \vdash_{\mathtt{M}} P : \tau \qquad \Gamma(u) = \mathrm{Gen}_{\Gamma}(\tau)}{\Gamma \vdash_{\mathtt{M}} \langle u \Leftarrow P \rangle : \omega} \text{ (decl +)}$$

$$\frac{\Gamma \vdash_{\mathtt{M}} P : \tau \qquad \Gamma(u) = \mathrm{Gen}_{\Gamma}(\tau) \prec \tau'}{\Gamma_{|u} \vdash_{\mathtt{M}} \mathrm{rec}\, u.P : \tau'} \text{ (rec +)}$$

which has been "implemented" in a version of ML, was shown to be undecidable in 1990. Henglein [18] and independently Kfoury, Tiuryn and Urzyczyn [22] shown that the typability problem in MM is log-space equivalent to semi-unification, where semi-unification is the problem of solving subsumption inequations between first order terms. Semi-unification has been shown to be recursively undecidable. Thus we can't reasonably expect typability for our calculus to be decidable, and we prove the undecidability of the type inference problem in Section 7.

To achieve decidable type reconstruction in $\pi^\star$, it is possible to draw one's inspiration from the Damas-Milner typing rule for the "let" construct. This provides us with new typing rule for declarations (decl) which contrasts with (decl +) in the same way that rules (rec) and (rec +) differ. Those new rules, as well as the derived rules for recursion, are presented in Table 6.

3.2 Example: a process-based cell

Just as there are more typable terms in the MM type inference system than in DM, there are strictly more typable processes with rule (decl +). We propose a practical example to illustrate this property.

A discussion on how to model cells in the blue calculus can be found in conclusion of [8]. Cells, readers may be more familiar with the term *reference*, used with a different meaning in this article, a classical example of mutable data structure encoded in process calculi. Its behaviour is very much like the single buffer of Section 2 except that accesses (messages on *get*) are not destructive. First, we introduce a new operator, $\langle a \Leftarrow P, b \Leftarrow Q \rangle$,

which combines two declarations in a single construct such that "resource fetching" becomes correlated. This operator, subsequently denoted by pair-declarations, verifies

$$\begin{cases} \langle a \Leftarrow P, b \Leftarrow Q\rangle \mid a \to P \\ \langle a \Leftarrow P, b \Leftarrow Q\rangle \mid b \to Q \end{cases} \tag{3}$$

This construction provides us with an encoding for records [9, 33], but it is a special kind of records since field extraction is not an application but a parallel composition. Thus it is asynchronous and non-deterministic. In fact, pair-declarations are better understood as "input-guarded choice", $\langle a \Leftarrow (\lambda\tilde{x})P, b \Leftarrow (\lambda\tilde{y})Q\rangle$ being operationally equivalent to the π-calculus process $a(\tilde{x}).P + b(\tilde{y}).Q$. However it is interesting to observe that the restriction on $\pi^\star$ that forbid abstraction on references is exactly the restriction in "ML-with-records" that field names cannot be abstracted. Using the notation for pair-declaration, a cell is the recursively defined process

$$cell =_{\text{def}} \text{rec}\, c.(\lambda v_0)\langle put \Leftarrow (\lambda x)(c\ x), get \Leftarrow (v_0 \mid (c\ v_0))\rangle$$

We invite the reader to consider the differences with "imperative" style references. In particular, non-determinism and behaviours arising from the asynchronous and call-by-name nature of blue calculus. As an exercise the reader might like to find how $(cell\ f \mid put\ g \mid get\ v_1 \ldots v_n)$ evaluates and why $((put\ get) \mid get)$ is a deadlocked process.

Starting from the λ-calculus style encodings for pairs: $[P, Q] =_{\text{def}} \lambda b.(b\ P\ Q)$, described in Section 2, one can devise an encoding for pair-declarations in the blue calculus. For the sake of brevity, we have indexed our translation with p, the "location" of the pair, and we suppose that this name is never captured.

$$\langle a \Leftarrow P, b \Leftarrow Q\rangle_p =_{\text{def}} \langle p \Leftarrow [P, Q]\rangle \mid \langle a \Leftarrow p\ T\rangle \mid \langle b \Leftarrow p\ K\rangle \tag{4}$$

Actually, this encodings is not faithful to Equation (3). References a and b must not appear in subject part of other declarations and we have the property

$$\begin{cases} \langle a \Leftarrow P, b \Leftarrow Q\rangle_p \mid a \to P \mid \langle b \Leftarrow p\ K\rangle \\ \langle a \Leftarrow P, b \Leftarrow Q\rangle_p \mid b \to Q \mid \langle a \Leftarrow p\ T\rangle \end{cases} \tag{5}$$

stating, in particular, that after a communication on a, $\langle a \Leftarrow P, b \Leftarrow Q\rangle_p$ is still responsive on b. This behaviour can be adjusted by replacing the definition of p in Equation (4), namely $\langle p \Leftarrow [P, Q]\rangle$, with the more complicated process $\langle p \Leftarrow ([P, Q] \mid \langle p \Leftarrow [a, b]\rangle)\rangle$. However, since a cell is always responsive on both references *put* and *get*, a quick argument shows that in the cell-example it is not necessary to change our definition.

It is possible to give results on the type of $\langle a \Leftarrow P, b \Leftarrow Q\rangle$ and ultimately of *cell*. In the monomorphic type system, $\Gamma \vdash_{\text{B}} P : \tau_p$ and $\Gamma \vdash_{\text{B}} Q : \tau_q$ implies $\Gamma \vdash_{\text{B}} [P, Q] : ((\tau_p \to \tau_q \to \alpha) \to \alpha)$. Since T (respectively K) has type $(\alpha \to \beta \to \alpha)$ (resp. $(\alpha \to \beta \to \beta)$).

Then $\langle a \Leftarrow P, b \Leftarrow Q\rangle$ is typable in $\vdash_{\mathtt{B}}$ iff τ_p and τ_q are equal. This constraint is different with a polymorphic type system. In particular we have the weaker requirement that a's type subsumes τ_p and b's type subsumes τ_q. Their actual value depends on the type discipline used.

For example, let us examine the *cell* process, where $P =_{\text{def}} (\lambda x)(c\ x)$ and $Q =_{\text{def}} (v_0 \mid (c\ v_0))$.

$$cell =_{\text{def}} \text{rec}\, c.(\lambda v_0)\left(\begin{array}{l}\langle p \Leftarrow (\nu ab)((\lambda f)(f\ a\ b) \mid \langle a \Leftarrow P\rangle \mid \langle b \Leftarrow Q\rangle)\rangle \\ \mid \langle put \Leftarrow p\ T\rangle \\ \mid \langle get \Leftarrow p\ K\rangle\end{array}\right)$$

In its recursive definition, reference c appears in both P and Q. In $\vdash_{\mathtt{M}}$, we can type this process assuming $c : \forall\alpha.(\beta \to \alpha)$. It follows that *cell* is typable with the hypothesis $put : \forall\alpha.(\beta \to \alpha)$. This is not possible in $\vdash_{\mathtt{D}}$, where, inside its definition, c must have a monomorphic type. Therefore, without polymorphic recursion, the cell process is typable with the stronger assumption that *put* have type $(\beta \to \beta)$. A consequence is that process $\langle u = (put\ 0 \mid P)\rangle$, which resets the cell value whenever a message on u has been consumed, is not typable in $\vdash_{\mathtt{D}}$ if P is not an integer expression, whereas, for example,

$$put : \forall\alpha.(\text{int} \to \alpha), get : \text{int}, u : \forall\alpha.\alpha \vdash_{\mathtt{M}} \langle u = (put\ 0 \mid (\lambda x)x)\rangle : \omega$$

Readers familiar with ML may be surprised. Indeed the naïve extension of the ML polymorphic type system to mutable structures is unsound and we just proved that cells are definable in $\pi^\star$ and can be typed. This difference in performance results from the difference in evaluation strategy between the two calculi. Evaluation in $\pi^\star$ is by name: it is not possible to reduce a process under a declaration, whereas evaluation in ML is by need: an expression defined in a let-construct is evaluated at most once and its result is shared. This good behaviour with respects to "imperative features" was already observed in the programming language QUEST [10] where polymorphically typed expression are evaluated each time their types are specialized. Another, more relevant, reference is MLN, the variant of ML with polymorphism by name described in [25].

3.3 General properties

Our type system is not equipped with the usual instantiation/generalization rules. In particular, processes are always given a simple type. We choose instead to give a syntax-oriented type system where these two operations are embedded in rules for instantiation (taut), application (app) and declaration (decl). Nonetheless we can easily extend our type system with generalization and instantiation and prove that both systems are equivalent. A natural consequence is that types given to processes can "implicitly" be considered polymorphic. Formally, the derived rule (sub) (Table 7), which states that the implicit meaning of sequent $\Gamma \vdash P : \tau$ is $\Gamma \vdash P : \text{Gen}_\Gamma(\tau)$, is sound. This property follows directly from (substitution) Lemma 4.3, used in the demonstration of the subject reduction property. Another standard

property that can be derived in our type system is (weak), which states, in particular, that weakening is valid.

Type System 7 Subtyping and weakening

$$\frac{\Gamma \vdash P : \tau \qquad \mathrm{Gen}_\Gamma(\tau) \prec \tau'}{\Gamma \vdash P : \tau'} \text{ (sub)} \qquad \frac{\Gamma \vdash P : \tau \qquad x \notin \mathrm{fn}(P)}{\Gamma_{|x}, x : \sigma \vdash P : \tau} \text{ (weak)}$$

A direct application of rule (sub) is the derived rule for process application

$$\frac{\Gamma \vdash P : \tau' \to \tau'' \qquad \Gamma \vdash Q : \tau \qquad \mathrm{Gen}_\Gamma(\tau) \prec \tau'}{\Gamma \vdash (P\ Q) : \tau''} \text{ (app)}$$

4 Subject reduction

A commonly expected property from type system for programming languages is that well-typed expressions remains well-typed after reduction. Moreover we expect to preserve typings. That is $(\Gamma \vdash P : \tau)$ and $(P \to Q) \Rightarrow \Gamma \vdash Q : \tau$. This property is called subject reduction. But, as pointed out by Boudol in [8, sec. 4], the "message sharing rule", i.e., relation $(P\ u \mid Q\ u) \ \rightharpoonup\ (P \mid Q)\ u$, is not compatible with an extension of the blue calculus to parametric polymorphism. That is, $(P\ u \mid Q\ u)$ may be typable even if $((P \mid Q)\ u)$ is not.

Example 4.1
We use the nil *type constant to represent idle processes or "silent" termination of operators. Suppose we have modeled a distributed application in which a message server, possibly connected with different databases, is able to print request results on a private medium. In* $\pi^\star$*, this server can be viewed as a simple replicated declaration* $\langle print = \ldots \rangle$ *with print of type* $\forall\alpha.\alpha \to \mathrm{nil}$*. We can define, in the same context, two specialized servers that applies their argument to the integer value* 1 *and the boolean value true* $\langle app2one = (\lambda f)(f\ 1) \rangle$ *and* $\langle app2true = (\lambda f)(f\ true) \rangle$ *whose respective types are* $(\mathrm{int} \to \mathrm{nil}) \to \mathrm{nil}$ *and* $(\mathrm{bool} \to \mathrm{nil}) \to \mathrm{nil}$*. Then, the process* $((app2one\ print) \mid (app2true\ print))$ *has type* nil*, while* $((app2one \mid app2true)\ print)$ *is not typable.*

We demonstrate the subject reduction property for a restricted version of the reduction, $\Longrightarrow$, such that structural equivalences is replaced by the weaker relation of heating $\rightharpoonup$ (see Section 2). In particular $\rightharpoonup$ is not symmetric, and thus is not an equivalence. However it is easy to be convinced that $\rightharpoonup$ is not necessary in reduction. Results proved in this section are for Mycroft-style type system, $\vdash_{\mathtt{M}}$, but are valid with $\vdash_{\mathtt{D}}$. Subsequently we denote typing sequent with $\vdash$ whenever a property is true for both system.

Theorem 4.1 (Subject reduction)
Reduction preserves typing

$$\Gamma \vdash P : \tau \text{ and } P \Longrightarrow Q \text{ implies } \Gamma \vdash Q : \tau$$

It is easy to show that typing is preserved by the heating relation.

Lemma 4.1
Heating preserves typing

$$\Gamma \vdash P : \tau \text{ and } P \rightharpoonup Q \text{ implies } \Gamma \vdash Q : \tau$$

Proof by induction on the structure of $P \rightharpoonup Q$. □

To prove the main theorem we need to demonstrate basic substitutions lemmas. That is properties between typings and substitution over processes and environments. Lemma 4.2 states that we can substitute a reference to a variable in a process whenever its type is more general. Lemma 4.3 states that typings are invariant under substitution of simple type to a type variable.

Lemma 4.2 (Substitution lemma (on references))
If $\Gamma, v : \sigma, x : \tau \vdash P : \tau'$, *with* x *variable and* $\sigma \prec \tau$, *then*

$$\Gamma, v : \sigma \vdash P[v/x] : \tau'$$

Lemma 4.3 (Substitution lemma (on type variables))
let α *be a type variable and* ω *be a simple type, then*

$$\Gamma \vdash P : \tau \Rightarrow \Gamma[\omega/\alpha] \vdash P : \tau[\omega/\alpha]$$

Proof (Lemma 4.2 and 4.3) The proofs of these lemmas are similar and use induction on the inference structure of sequent $\Gamma, v : \sigma, x : \tau \vdash P : \tau'$ or $\Gamma \vdash P : \tau$. The details are self-evident. Lemma 4.2 is less standard than Lemma 4.3 and is only used in the case for rule (β) in the following proof. It is interesting to remark that this lemma is not necessary if we replace β-reduction by the rule

$$((\lambda x)P\ u) \rightarrow (\nu x)(P \mid \langle x = u \rangle)) \qquad (v \text{ fresh})$$

inspired by Equation (1) presented in page 4 □

We can demonstrate the result stated at the beginning of this section

Proof (Theorem 4.1) we use induction on the inference $P \Longrightarrow Q$.

case (struct): the result follows directly from Lemma 4.1 and the induction hypothesis.

case (app): By assumption we have that $P\ u \Longrightarrow Q\ u$ with $P \Longrightarrow Q$ and $\Gamma \vdash (P\ u) : \tau$. Therefore, it must be the case that

$$\frac{\Gamma \vdash P : \tau' \to \tau \qquad \Gamma(u) = \sigma \prec \tau'}{\Gamma \vdash P\ u : \tau}\ (\text{app})$$

We have $\Gamma \vdash P : \tau' \to \tau$, then, using induction, we have that $\Gamma \vdash Q : \tau' \to \tau$. Applying rule (app) it follows that $\Gamma \vdash (Q\ u) : \tau$. Case (par) and (new) are similar.

case (β): the last reduction step is $(\lambda x)P\ u \to P[u/x]$. It must be the case that the last rules used to infer sequent $\Gamma \vdash (\lambda x)P\ u : \tau$ are

$$\frac{\dfrac{\Gamma, x : \tau' \vdash P : \tau}{\Gamma \vdash (\lambda x)P : \tau' \to \tau}\ (\text{abs}) \qquad \Gamma(u) = \sigma \prec \tau'}{\Gamma \vdash (\lambda x)P\ u : \tau}\ (\text{app})$$

$\Gamma, x : \tau' \vdash P : \tau$ and $\Gamma(u) = \sigma \prec \tau'$, therefore, using Lemma 4.2, it follows that $\Gamma \vdash P[u/x] : \tau$.

case (ρ): By assumption, we have that the last reduction step is $(u \mid \langle u \Leftarrow P \rangle) \to P$. Therefore, the derivation for $\Gamma \vdash (u \mid \langle u \Leftarrow P \rangle) : \tau$ is

$$\frac{\dfrac{\Gamma \vdash P : \tau' \qquad \Gamma(u) = \text{Gen}_\Gamma(\tau')}{\Gamma \vdash \langle u \Leftarrow P \rangle : \tau}\ (\text{decl}+) \qquad \dfrac{\Gamma(u) = \sigma \prec \tau}{\Gamma \vdash u : \tau}\ (\text{taut})}{\Gamma \vdash (u \mid \langle u \Leftarrow P \rangle) : \tau}\ (\text{par})$$

We have that $\text{Gen}_\Gamma(\tau') \prec \tau$, therefore it exists a substitution S over variables from $\text{fn}(\tau') \setminus \text{fn}(\Gamma)$ such that $S\tau' = \tau$. Using Lemma 4.3, $S\Gamma \vdash P : S\tau'$ and, as $\text{fn}(\Gamma) \cap \text{domain}(S) = \emptyset$, $S\Gamma = \Gamma$. It follows that $\Gamma \vdash P : \tau$.

□

To conclude, let us give a simple example that demonstrates the need to forbid reference abstraction in the subject part of a declaration. Let R be the (invalid) process

$$R =_{\text{def}} \langle u \Leftarrow P \rangle \mid ((\lambda x)\langle x \Leftarrow Q \rangle\ u) \qquad (u \text{ not in } P \text{ or } Q)$$

and Γ be a context such that $\Gamma(u) = \sigma$, $\Gamma \vdash P : \tau_p$ and $\Gamma \vdash Q : \tau_q$, e.g.,

$$Q = (\lambda x)(+\ x\ 1),\ \tau_q = \text{int} \to \text{int},\ P = (\lambda x)x,\ \tau_p = \alpha \to \alpha \text{ and } \sigma = \forall \alpha.\tau_p$$

For R to be well-typed, it is sufficient that $\sigma = \text{Gen}_\Gamma(\tau_p)$ (rule (decl +)) and $\sigma \prec \tau_q$ (rules (decl +), (abs) and (app)). R reduces in one β-reduction step to

$$R \to R' = \langle u \Leftarrow P \rangle \mid \langle u \Leftarrow Q \rangle = \langle u \Leftarrow (\lambda x)x \rangle \mid \langle u \Leftarrow (\lambda x)(+\ x\ 1) \rangle$$

The conditions required to type R' in Γ are stronger, since we must have $\sigma = \text{Gen}_\Gamma(\tau_p) = \text{Gen}_\Gamma(\tau_q)$, constraints which are trivially false.

This problem does not arise in system $\vdash_{\mathtt{B}}$, i.e., the monomorphic type system is sound in the blue-calculus without restriction on reference abstraction. We are in a situation quite comparable to the naïve extension of the ML polymorphic type system to references.,and in fact, to all imperative constructs. Many type systems, have been proposed for typing polymorphic references: Tofte [34], Mac Queen, Wright and Leroy ([24] gives a good overview). It would be interesting to study how those systems can be adapted to $\pi^\star$. An intuition is given in Appendix A.

5 Principal types

Beside subject reduction, an important property for type systems is the *principal type* property, which states that for any process P and context Γ, such that P is typable in Γ, there is a type that represents all of the possible typings of P in Γ.

Theorem 5.1 (Principal type)
If P is typable in the context Γ (i.e., $\exists \tau', \Gamma \vdash_{\mathtt{M}} P : \tau'$ is derivable), there exists a "more general" type scheme σ such that for every derivable sequent $\Gamma \vdash_{\mathtt{M}} P : \tau \Leftrightarrow \sigma \prec \tau$.

Proof See Appendix B. □

We extend the subsumption relation $\prec$ to type schemes and typing environments

$$\sigma \prec \rho \quad \text{iff} \quad \forall \tau, \rho \prec \tau \Rightarrow \sigma \prec \tau$$

$$\Delta \prec \Gamma \quad \text{iff} \quad \forall (u : \sigma) \in \Gamma, \exists (u : \rho) \in \Delta \text{ and } \rho \prec \sigma$$

the sequent $\Gamma \vdash P : \tau$ is said to be more general than $\Delta \vdash P : \omega$ if there is a substitution S such that $S(\Delta) \prec \Gamma$ and $\mathrm{Gen}_\Gamma(\tau) \prec S(\mathrm{Gen}_\Delta(\omega))$. The intuition is that the most general typing (if it exists) expects less from the context and provides more. With this definition, it happens that a "most general typing" can't be given in every case. More specifically, we have that $\vdash_{\mathtt{M}}$ lacks the *principal typing* property [20].

Example 5.1
$a : \forall\alpha.\alpha \vdash (a\ a) : \beta$ *and* $a : \forall\alpha.(\alpha \to \alpha) \vdash (a\ a) : \beta \to \beta$

But we can prove that for every typable process P, there is a particular typing which represent all the possible typing of P. A property comparable for ML to Theorem 1.5.3 in [11].

Theorem 5.2
For every typable process P there exists a most general typing $\Gamma_p \vdash P : \tau_p$ such that: if $\Delta \vdash P : \omega$, there is a substitution S such that $\mathrm{Gen}_{\Gamma_p}(\tau_p) \prec S(\mathrm{Gen}_\Delta(\omega))$ and $\Gamma_p \prec S(\Delta_{|\mathrm{fn}(P)})$.

Table 2 translation from ML to $\pi^\star$ $[\![.]\!] : \mathrm{ML} \mapsto \pi^\star$

$$
\begin{array}{rcll}
[\![x]\!] & = & x & \\
[\![\lambda x.M]\!] & = & (\lambda x)[\![M]\!] & \\
[\![M\ N]\!] & = & (\nu v)([\![M]\!]\ v \mid \langle v = [\![N]\!]\rangle) & (v \text{ fresh}) \\
 & = & \mathrm{def}\,\langle v = [\![N]\!]\rangle \,\mathrm{in}\, ([\![M]\!]\ v) & \\
[\![\mathrm{letrec}\, u_1 = N_1 \text{ and } \ldots \mathrm{in}\, M]\!] & = & (\nu \tilde{u})([\![M]\!] \mid \langle u_1 = [\![N_1]\!]\rangle \mid \ldots) & \\
 & = & \mathrm{def}\,\langle u_1 = [\![N_1]\!]\rangle \mid \ldots \text{ in } [\![M]\!] &
\end{array}
$$

Proof The proof is a direct consequence of results shown in Appendix B. For example, using the same notation as in Theorem 5.1, we associate with the process of Example 5.1 the vector $(\forall\alpha.\alpha, \forall\alpha.\alpha)$, i.e., the typing $a : \forall\alpha.\alpha \vdash (a\ a) : \alpha$.

6 Relationship with ML

In ML, recursive definitions are only used in the definition part of a let-construct. This yields a derived construct, letrec, which is a shorthand for

$$\mathrm{letrec}\, x = N \,\mathrm{in}\, M \ =_{\mathrm{def}}\ \mathrm{let}\, x = \mathrm{rec}\, x.N \,\mathrm{in}\, M$$

Practically, the letrec construct is a more general operator which allows mutually recursive function declarations in ML ($\mathrm{letrec}\, u_1 = N_1$ and $\ldots \mathrm{in}\, M$).

Definition 6.1 ($\mathrm{def}\, J \,\mathrm{in}\, P$)
We distinguish a new syntactic category in the grammar of $\pi^\star$, namely declaration bodies, which are the parallel composition of (linear or replicated) declarations $B ::= D \mid \cdots \mid D$. The def-construct, $\mathrm{def}\, B \,\mathrm{in}\, P$, is a $\pi^\star$ derivable operator used as shorthand for

$$\mathrm{def}\, B \,\mathrm{in}\, P \ =_{\mathrm{def}}\ (\nu \tilde{u})(B \mid P) \quad \textit{where } \tilde{u} = \mathrm{decl}(B)$$

with the restriction that $\mathrm{decl}(B) \cap \mathrm{decl}(P) = \emptyset$.

Side condition $\mathrm{decl}(B) \cap \mathrm{decl}(P) = \emptyset$ ensure that there is no redefinition in P of references defined in B. That is, there is no declarations in P with subject part in $\mathrm{decl}(B)$. This property is preserved by reduction since, although the scope of restricted declarations can be widened (this is the classical scope extrusion mechanism of π-calculus), no declarations can be dynamically created. Table 2 gives a simple translation from ML (with mutually recursive declarations) to the subset of the blue calculus in which declarations are replaced with def-operators.

Derived typing rules for the def-construct can be given in both our Damas and Mycroft like type system but, drawing one's inspiration from ML typing rules for the letrec construct

(see Table 8), we define a new type inference system $\vdash_{\forall \mathtt{B}}$ such that we restrict type polymorphism to occur only in the def-construct. Rules for this new system are given in Table 9 with the derived rule for the def-construct in $\vdash_{\mathtt{M}}$. It is interesting to compare this new type system with the implicit typing à la ML devised for the join-calculus [16], where authors ran across the same classical limitation of typing for mutually-recursive functions.

Type System 8 Typing rules for letrec in ML

$$\frac{\Gamma \cup \{u_i : \mathrm{Gen}_{\Delta_i}(\tau_i)\} \vdash_{\mathtt{M}} N_i : \tau_i \qquad \Gamma \cup \{u_i : \mathrm{Gen}_{\Delta_i}(\tau_i)\} \vdash_{\mathtt{M}} M : \tau}{\Gamma \vdash_{\mathtt{M}} \mathrm{letrec}\, u_1 = N_1 \text{ and } \ldots \mathrm{in}\, M : \tau} \text{ (letrec +)}$$

$$\frac{\Gamma \cup \{u_i : \tau_i\} \vdash_{\mathtt{D}} N_i : \tau_i \qquad \Gamma \cup \{u_i : \mathrm{Gen}_{\Delta_i}(\tau_i)\} \vdash_{\mathtt{D}} M : \tau}{\Gamma \vdash_{\mathtt{D}} \mathrm{letrec}\, u_1 = N_1 \text{ and } \ldots \mathrm{in}\, M : \tau} \text{ (letrec)}$$

(where $\Delta_i = \Gamma \cup \{u_j : \tau_j \mid j \neq i\}$)

Type System 9 $\vdash_{\forall \mathtt{B}}$, a new polymorphic type system for $\pi^\star$

$$\frac{\Gamma \cup \{u_i : \mathrm{Gen}_{\Delta_i}(\tau_i)\} \vdash_{\mathtt{M}} R_i : \tau_i \qquad \Gamma \cup \{u_i : \mathrm{Gen}_{\Delta_i}(\tau_i)\} \vdash_{M\,M} P : \tau}{\Gamma \vdash_{\mathtt{M}} \mathrm{def}\,(\langle u_i \Leftarrow R_i \rangle) \,\mathrm{in}\, P : \tau} \text{ (def +)}$$

$$\frac{\Gamma, u : \tau \vdash_{\forall \mathtt{B}} P : \tau}{\Gamma, u : \tau \vdash_{\forall \mathtt{B}} \langle u = P \rangle : \omega} \text{ (decl -)}$$

$$\frac{\Gamma \cup \{u_i : \tau_i\} \vdash_{\forall \mathtt{B}} R_i : \tau_i \qquad \Gamma \cup \{u_i : \mathrm{Gen}_{\Delta_i}(\tau_i)\} \vdash_{\forall \mathtt{B}} P : \rho}{\Gamma \vdash_{\forall \mathtt{B}} \mathrm{def}\,(\langle u_i \Leftarrow R_i \rangle) \,\mathrm{in}\, P : \rho} \text{ (def -)}$$

(where $\Delta_i = \Gamma \cup \{u_j : \tau_j \mid j \neq i\}$)

It is possible to demonstrate a result equivalent to the one presented by Boudol in [8, Proposition 4.2]. Let $\Gamma \vdash M : \tau$ [ML] denote sequents from the ML type system (see table 4 on page 11). In the same way, let $\Gamma \vdash P : \tau$ [$\pi^\star$] denote sequents inferred using the polymorphic type system for $\pi^\star$.

Theorem 6.1 (Equivalence on typings)
If $\Gamma \vdash_{\mathtt{D}} M : \tau$ [ML] *then* $\Gamma \vdash_{\forall \mathtt{B}} [\![M]\!] : \tau$ [$\pi^\star$]. *Conversely if* $\Gamma \vdash_{\forall \mathtt{B}} [\![M]\!] : \tau$ [$\pi^\star$] *then* $\Delta \vdash_{\mathtt{D}} M : \tau$ [ML] *with* $\Delta = \Gamma_{|(\mathrm{fn}(\Gamma) \setminus \mathrm{fn}(M))}$. *This property is true with* $\vdash_{\mathtt{M}}$ [ML] *and* $\vdash_{\mathtt{M}}$ [$\pi^\star$].

7 Type reconstruction complexity

As stated in Section 3.1, the typability in MM type inference system has been proved undecidable [22, 18]. Therefore, an immediate consequence of Theorem 6.1 is

Theorem 7.1
The type inference problem for $\vdash_{\mathtt{M}} [\pi^\star]$ *is undecidable.*

In fact, we can prove a finer property quoting F. Henglein in [18]: "*for every system of inequations* $\mathcal{I}$*, there is a log-space computable expression of the form* $\mathrm{rec}\, x.M$ *where* e *is let and rec-free,i.e., e is a pure* λ*-term, such that* $\mathcal{I}$ *is semiunifiable if and only if* $\mathrm{rec}\, x.M$ *is MM typable*". Another reduction is given by A. J. Kfoury *et al.* in [22] with the same characteristics. As the translation from ML-term to $\pi^\star$-term preserve typings (Theorem 6.1), we obtain a reduction from semi-unification to typabilty in $\vdash_{\mathtt{M}} [\pi^\star]$. Moreover,

Lemma 7.1 (semi-unification reduce to typability in $\vdash_{\mathtt{M}}$)
for every system of inequations $\mathcal{I}$*, there is a log-space computable expression of the form* $\mathrm{rec}\, x.P$ *such that* P *has no recusive definition (i.e., no declarations* $\langle u = Q\rangle$ *with* $u \in \mathrm{fn}(Q)$*) and that* $\mathcal{I}$ *is semi-unifiable if and only if* $\mathrm{rec}\, x.P$ *is MM typable.*

Proof for Theorem 7.1 follows since semi-unification has been shown recursively undecidable [21, 36]. We show in fact that typability in the Damas-like type inference system $\vdash_{\mathtt{D}}$ is also undecidable

Theorem 7.2
The type inference problem for $\vdash_{\mathtt{D}} [\pi^\star]$ *is undecidable.*

Proof To prove Theorem 7.2, it is sufficient to prove that the semi-unification problem reduces to typability in $\vdash_{\mathtt{D}}$. Suppose there's a typing judgment $\Gamma \vdash_{\mathtt{M}} \mathrm{rec}\, u.a_u : \rho$ such that there is no recursive declaration in a_u. We first prove that[4] $\Gamma \vdash_{\mathtt{D}} (\nu u)(\mathrm{def}\, v = u \,\mathrm{in}\, (\langle u = a_v\rangle \mid v)) : \rho$. Let v be a new reference and $a_v = a_u[v/u]$:

$$\Gamma \vdash_{\mathtt{D}} (\nu u)(\mathrm{def}\, v = u \,\mathrm{in}\, (\langle u = a_v\rangle \mid v)) : \rho$$

from the relation $\Gamma \vdash_{\mathtt{M}} \mathrm{rec}\, u.a_u : \rho$ and the rule given in table 6 on page 12, we deduce that there exists a type τ such that (we use σ for $\mathrm{Gen}_\Gamma(\tau)$)

$$\frac{\Gamma, u : \sigma \vdash_{\mathtt{M}} a_u : \tau \qquad \sigma \prec \rho}{\Gamma \vdash_{\mathtt{M}} \mathrm{rec}\, u.a_u : \rho}\ (\mathrm{rec}\ +)$$

By hypothesis, a_v has no recursive definitions (it is even sufficient to suppose that a_v is the translation of a pure λ-term), then $\forall \tau, \Gamma \vdash_{\mathtt{M}} a_u : \tau \Rightarrow \Gamma \vdash_{\mathtt{D}} a_u : \tau$. Moreover, v is fresh. Then $\Gamma, v : \sigma \vdash_{\mathtt{D}} a_u : \tau$ (weakening) and $\Gamma, v : \sigma \vdash_{\mathtt{D}} a_v : \tau$ (same as lemma 4.2 on page 16).

[4] $\mathrm{def}\, v = u \,\mathrm{in}\, (\langle u = a_v\rangle \mid v) =_{\mathrm{def}} (\nu v)(\langle v = u\rangle \mid \langle u = a_v\rangle \mid v)$

$$\dfrac{\dfrac{\dfrac{\Gamma, v : \sigma \vdash_{\mathtt{D}} a_v : \tau \qquad u \notin \mathrm{fn}(a_v)}{\Gamma, u : \tau, v : \sigma \vdash_{\mathtt{D}} a_v : \tau}\ \text{(weakening)}}{\Gamma, u : \sigma, v : \sigma \vdash_{\mathtt{D}} \langle u = (a\ v)\rangle : \rho}\ \text{(decl)} \qquad \sigma \prec \rho}{\Gamma, u : \sigma, v : \sigma \vdash_{\mathtt{D}} \langle u = a_v\rangle \mid v : \rho}\ \text{(par)}$$

It is also the case that

$$\dfrac{\dfrac{\dfrac{\sigma \prec \tau}{\Gamma, u : \sigma \vdash_{\mathtt{D}} u : \tau}\ \text{(taut)} \qquad v \text{ fresh}}{\Gamma, u : \sigma, v : \tau \vdash_{\mathtt{D}} u : \tau}}{\Gamma, u : \sigma, v : \mathrm{Gen}_\Gamma(\tau) \vdash_{\mathtt{D}} \langle v = u\rangle : \rho}\ \text{(decl)}$$

Then, applying rules (par) and (new):

$$\Gamma \vdash_{\mathtt{D}} (\nu u)(\mathrm{def}\, v = u \,\mathrm{in}\, (\langle u = a_v\rangle \mid v)) : \rho$$

using Lemma 7.1, it follows that for every system of inequations $\mathcal{I}$, we can exhibit a $\pi^\star$ process P such that $\mathcal{I}$ is semi-unifiable iff P is typable in the system $\vdash_{\mathtt{D}}$. □

8 Conclusion

We have presented, in the present paper, a type system for $\pi^\star$ with polymorphic recursion adapted from the ML typing discipline, and we proved it enjoys subject reduction and the principal type property.

These are interesting results for the blue calculus as a programming notation for higher-order concurrency. But a type system is hardly helpful if one cannot gives an algorithm that can tell whether or not a program is type-correct. A first approach will be to restrict our type discipline. This is the intention behind the introduction of system $\vdash_{\forall\mathtt{B}}$. The type reconstruction problem in $\vdash_{\forall\mathtt{B}}$ trivially reduces to first order-unification and, consequently, is decidable: it is possible, for example, to adapt the deterministic type inference algorithm given in [14]. Other approaches use the more expressive system $\vdash_{\mathtt{M}}$. The simplest solution is to provide explicit type declarations for every reference. One can also devise specific type inference algorithms [29] or an algorithm based on semi-algorithms for semi-unification [18, sec. 5]: we trade off expressiveness for a type inference algorithm guaranteed to finish only for well-typed programs.

It is also interesting to understand why those algorithms are more complicated than for the "ML with polymorphic recursion" case. Contrary to the ML let-construct (and the join of [15]), the scope of declarations are not syntactically fixed in the blue calculus. Therefore, if we think of $\pi^\star$ in terms of programming languages and of declarations in terms of function definitions, we are in the case where all function declarations are top-level and mutually recursive. Moreover, a reference can appear in subject part of many declarations, a situation comparable to multiply defined functions. This problem is called *incremental polymorphic type checking with update* by Mycroft in [30] and It is exactly the case of the programming

language PROLOG where predicates are incrementally and mutually defined. Type systems for PROLOG have been extensively studied [31] and it should be possible to adapt them to our need.

An other interesting contribution of this work are the consequences we can draw on the π-calculus. Simple translation exists between the blue calculus and the asynchronous π calculus [4, 19]. Therefore, one can easily deduce that it is not possible to devise an implicit type system "à la ML" for the π-calculus along the lines of $\vdash_{\mathtt{M}}$. Indeed, the same arguments applies in each case. For the same reason, one can easily derive a polymorphic type system for π_a^i, Michele Boreale's calculus [3], from our type system for the restricted version of $\pi^\star$ used in this article, i.e., without abstraction on references. In fact, at the cost of simple transformation, it is also possible to type πI, the sub-language of the π-calculus such that only private names can be exchanged among processes.

Acknowledgments

Thanks for comments from Didier Rémy who suggested that typability in $\vdash_{\mathtt{D}}$, the natural adaptation of the Damas-Milner type system to the blue calculus, was undecidable. I am also thankful to Julian Rathke and Gérard Boudol for valuable comments on this article.

A Polymorphism and higher-order references

An approach, commonly taken in ML to constrain polymorphism in references, consists in tagging type variables that appears free in a reference type. Those variables are often called weak or dangerous. Generalization is then redefined to make special treatments of weak variables. The simplest, but over-restrictive, approach consists in forbidding generalization of weak variables.

A weak variables is denoted α^*. We extend this notation to types $((\tau' \to \tau)^* =_{\text{def}} \tau'^* \to \tau^*)$. Drawing a parallel with ML, we redefine the type system to weaken potentially dangerous generic type variables. In particular we have to consider a rule to type abstraction on references

$$\frac{\Gamma \vdash P : \tau \qquad \Gamma(u) = \tau'^*}{\Gamma_{|x} \vdash (\lambda u)P : \tau'^* \to \tau} \text{ (abs } *)$$

Remember that with our conventions, u is a reference and τ' is a simple type. Then, rule (abs $*$) allows abstraction on u with the classical limitation that u's type is not a type scheme. Otherwise we have to consider system F types such that quantification is not restricted to be top-level.

We also redefine the subsumption relatiion $\prec$ to be such that $\forall\alpha.\alpha \not\prec \tau^*$. More formally, we forbid weak variables in generalization of type schemes, i.e., $\forall\tilde{\alpha}.\tau \prec \omega$ iff exist some simple

types $(\tilde{\tau_i})_{1\leq i\leq n}$ such that $\omega = \tau[\tilde{\tau_i}/\tilde{\alpha}]$ and the τ_i's have no weak variables. Let us see what happens with the example given Section 4

$$R =_{\text{def}} \langle u \Leftarrow (\lambda x)x\rangle \mid ((\lambda x)\langle x \Leftarrow (\lambda y)(+\ y\ 1)\rangle\ \ u)$$

Using rule (abs $*$), it follows that

$$\begin{array}{l} x : (\text{int} \to \text{int})^* \vdash \langle x \Leftarrow (\lambda y)(+\ y\ 1)\rangle : \omega \\ \quad\Longrightarrow\quad \emptyset \vdash (\lambda x)\langle x \Leftarrow (\lambda y)(+\ y\ 1)\rangle : (\text{int} \to \text{int})^* \to \omega \end{array}$$

With the new definition for subsumption, $\forall\alpha.(\alpha \to \alpha) \not\prec \text{int}^* \to \text{int}^*$. Therefore the unique valid typing is $u : \text{int}^* \to \text{int}^* \vdash R : \omega$, which is preserved after reduction.

We have only sketched out a polymorphic type system for the original blue calculus, i.e., without restrictions on references. But there is enough subtleties in the type discipline for the calculus studied in this paper to reserve a full treatment for ongoing research.

B Principal type

This appendix contains a proof for the principal type property, Theorem 5, stated in the paper.

B.1 Type semantics

Let $\mathcal{T}$ be the sets of all monomorphic and polymorphic types augmented with a greatest element $\top$, the *error* types, such that $\forall\sigma \in \mathcal{T}, \sigma \prec \top$. Theorem B.1 is a well known result. It appears, for example, in [29].

Theorem B.1
$(\mathcal{T}, \prec)$ is a partial order with least element $\forall\alpha.\alpha$, greatest element $\top$ and such that any subset $X \subset \mathcal{T}$ has a greatest lower bound $\bigsqcap X$.

Let us briefly explain the intuition behind the use of partial order theory. We denote by $\mathcal{T}(\Gamma, P)$ the set of P's type in the environment Γ

$$\mathcal{T}(\Gamma, P) = \left\{\sigma \,\middle|\, \Gamma \vdash P : \sigma\right\}$$

and we consider the slightly modified type system where rules for instantiation and generalization are added.

Type System 10 generalization and instantiation

$$\frac{\Gamma \vdash P : \sigma \qquad \alpha \notin \text{fn}(\Gamma)}{\Gamma \vdash P : \forall\alpha.\sigma}\ (\text{gen}) \qquad\qquad \frac{\Gamma \vdash P : \sigma \qquad \sigma \prec \rho}{\Gamma \vdash P : \rho}\ (\text{inst})$$

In this new (equivalent) type system, rule (gen) can be used at the end of a typing to give a type scheme to a process. This is reminiscent of our discussion on rule (sub) in Section 3.3. Properties of our type system can now be rephrased in this formal framework. For example, using Lemma 4.3, we have the property that $\mathcal{T}(\Gamma, P)$ is an *ideal*

$$\sigma_1 \in \mathcal{T}(\Gamma, P) \wedge \sigma_1 \prec \sigma_2 \Rightarrow \sigma_2 \in \mathcal{T}(\Gamma, P)$$

Likewise, the principal type property is equivalent to

$$\phi = \bigsqcap \mathcal{T}(\Gamma, P) \text{ is in } \mathcal{T}(\Gamma, P) \tag{6}$$

since, by definition of the greatest lower bound, $\forall \sigma, \Gamma \vdash P : \sigma \Rightarrow \phi \prec \sigma$. Such ideal are called *principal*. In fact, it is sufficient to prove the weaker relation

$$\left\{ \begin{array}{l} \Gamma \vdash P : \sigma_1 \\ \Gamma \vdash P : \sigma_2 \end{array} \right. \Rightarrow \Gamma \vdash P : \sigma_1 \sqcap \sigma_2 \tag{7}$$

As we are interested in results for the syntax directed type system $\vdash_{\mathtt{M}}$, we demonstrate instead the equivalent equation

$$\left\{ \begin{array}{l} \Gamma \vdash_{\mathtt{M}} P : \tau_1 \\ \Gamma \vdash_{\mathtt{M}} P : \tau_2 \\ \forall \tilde{\alpha}.\tau = \tau_1 \sqcap \tau_2 \end{array} \right. \Rightarrow \Gamma \vdash_{\mathtt{M}} P : \tau \tag{8}$$

B.2 Cartesian product over types

We introduce some notations necessary for the proof of Equation (8). Let $\mathcal{T}_n$ denotes the set of type vectors of size n, or rather type schemes vectors

$$\begin{array}{rcl} \Pi & ::= & (\tau_i)_{1 \leq i \leq n} \\ \Phi & ::= & (\sigma_i)_{1 \leq i \leq n} \\ \Psi & ::= & \Phi \mid \forall \alpha.\Psi \end{array}$$

We use ϕ to range over Φ and ψ over universally quantified vectors (element of Ψ). Subsumption is defined like $\prec$ on $\mathcal{T}$. Remark that $\prec_n$ is not the usual "product order" $\prec^n =_{\text{def}} (\prec \times \cdots \times \prec)$ such that $\psi_1 \prec^n \psi_2 \Leftrightarrow \forall i.(\psi_1^i \prec \psi_2^i)$

$$\left\{ \begin{array}{l} \tilde{\tau} \text{ simple types} \\ \forall i \quad \sigma_i[\tilde{\tau}/\tilde{\alpha}] \prec \rho_i \end{array} \right. \Rightarrow \forall \tilde{\alpha}.(\sigma_i)_{1 \leq i \leq n} \prec_n (\rho_i)_{1 \leq i \leq n}$$

this definition is extended to quantified type vectors

$$\psi_1 \prec_n \psi_2 \text{ iff } \forall \phi, \psi_2 \prec_n \phi \Rightarrow \psi_1 \prec_n \phi$$

Regarding the definition of type schemes given Section 3, it would have been more natural to consider quantified vectors of simple types instead of type schemes vector, i.e., we could

have replaced Φ with Π in definition of Ψ. In fact, each element of $\mathcal{T}_n$ has an equivalent in this form, e.g.,

$$(\forall\alpha.\alpha, \forall\alpha.(\beta \to \alpha)) \sim_2 \forall\alpha_0\alpha_1.(\alpha_0, (\beta \to \alpha_1))$$

where $\sim$ denotes the (classical) equivalence such that $\sigma \sim \rho \Leftrightarrow \sigma \prec \rho$ and $\rho \prec \sigma$. This is equivalent to writing a type in its "canonical form" (remark that here, the representant is not unique).

Theorem B.2
$(\mathcal{T}_n, \prec_n)$ is a partial order with least element $(\forall\alpha.\alpha)_{1\leq i\leq n}$, greatest element $(\top)_{1\leq i\leq n}$ and such that any subset $X \subset \mathcal{T}_n$ has a greatest lower bound $\bigsqcap X$.

Proof Let ψ_1 and ψ_2 be in $\mathcal{T}_n$. We consider their representations in "canonical form": $\psi_1 = \forall\tilde{\alpha}.(\tau_i)_{1\leq i\leq n}$ and $\psi_2 = \forall\tilde{\delta}.(\omega_i)_{1\leq i\leq n}$. It is easy to prove that subsumption between type vectors is related to subsumption between types following the equation

$$\psi_1 \prec_n \psi_2 \quad \Leftrightarrow \quad \forall\tilde{\alpha}.(\tau_1 \to \cdots \to \tau_n) \prec \forall\tilde{\delta}.(\omega_1 \to \cdots \to \omega_n)$$

In particular

$$\begin{aligned} \forall\tilde{\alpha}.(\tau_1 \to \cdots \to \tau_n) &= \textstyle\bigsqcap_{1\leq j\leq k} \left(\forall\tilde{\delta^j}.(\omega_1^j \to \cdots \to \omega_n^j)\right) \\ \Rightarrow\ \forall\tilde{\alpha}.(\tau_i)_{1\leq i\leq n} &= \textstyle\bigsqcap_{1\leq j\leq k} \left(\forall\tilde{\delta^j}.\left(\omega_i^j\right)_{1\leq i\leq n}\right) \end{aligned}$$

□

Remark that $\mathcal{T}_n$ is not equivalent to the sets of type schemes vectors, e.g., the vector $\forall\alpha.(\alpha, \alpha)$ is in $\Psi \setminus \Phi$. Let us denote $D((\sigma_i)_{1\leq i\leq n})$ the set of type variables appearing free in more than one σ_i

$$D((\sigma_i)_{1\leq i\leq n}) =_{\text{def}} \left\{\alpha \;\middle|\; \exists i \neq j, \alpha \in \text{fn}(\sigma_i) \text{ and } \alpha \in \text{fn}(\sigma_j)\right\}$$

We suppose, for the rest of the paper, that type vectors are given in the "second canonical form"

$$\Psi ::= \forall\tilde{\alpha}.(\sigma_i)_{1\leq i\leq n} \quad \text{with } \tilde{\alpha} \subset D((\sigma_i)_{1\leq i\leq n})$$

B.3 Proof

We could associate to each environment Γ ($\Gamma =_{\text{def}} x_1 : \sigma_1, \ldots, x_n : \sigma_n$) the type vector $\gamma =_{\text{def}} (\sigma_1, \ldots, \sigma_n) \in \Phi$, and to each typing judgment $\Gamma \vdash_{\mathtt{M}} P : \sigma$, the vector $(\sigma_1, \ldots, \sigma_n, \sigma) \in \Phi$, also denoted (γ, σ). Inversely, we associate to $\forall\tilde{\alpha}.(\gamma, \sigma)$ the typing $\Gamma \vdash_{\mathtt{M}} P : \tau$ such that $\sigma = \forall\tilde{\delta}.\tau$ and $\tilde{\delta} \cap \text{fn}(\gamma) = \emptyset$. Remember that, with our convention, $\tilde{\alpha} \subset D((\gamma, \sigma))$. Since we

are not allowed to give type scheme to variables, the definition of Γ is not immediate. Let σ_i be the i^{th} component of γ. If x_i is a reference, $(x_i : \sigma_i) \in \Gamma$, else, let $\sigma_i = \forall\tilde{\eta}.\omega$ ($\tilde{\eta}$ fresh variables), $(x_i : \omega) \in \Gamma$.

Equation (8) is a direct sub-case of Equation (9) such that $\Gamma_1 = \Gamma_2 = \Gamma$

$$\left\{\begin{array}{l}\Gamma_1 \vdash_{\mathtt{M}} P : \tau_1\\ \Gamma_2 \vdash_{\mathtt{M}} P : \tau_2\\ \forall\tilde{\alpha}.(\gamma,\sigma) = (\gamma_1,\tau_1) \sqcap (\gamma_2,\tau_2)\\ \sigma = \forall\tilde{\delta}.\tau\end{array}\right. \Rightarrow \Gamma \vdash_{\mathtt{M}} P : \tau \qquad (9)$$

To demonstrate Equation (9) we need some basic properties on the greatest lower bound. The proof of Lemma B.1 is a trivial adaptation of the argument in proof of Theorem B.2.

Lemma B.1

1. $\forall\tilde{\alpha}.(\sigma_i)_{1\le i\le n} = \prod_j \left(\forall\tilde{\alpha}^j.\left(\sigma_i^j\right)_{1\le i\le n}\right) \Rightarrow \forall\tilde{\alpha}.\sigma_i \sim \prod_j \left(\forall\tilde{\alpha}^j.\sigma_i^j\right)$
2. $\forall\tilde{\alpha}.(\gamma,\sigma) = \prod_i(\gamma_i,\tau_1^i \to \tau_2^i) \Rightarrow \left\{\begin{array}{l}\exists(\tau_1,\tau_2).\ \sigma = \forall\tilde{\delta}.(\tau_1 \to \tau_2)\\ \forall\tilde{\alpha}.(\gamma,\forall\tilde{\delta}.\tau_2) = \prod_i(\gamma_i,\tau_2^i)\\ \forall\tilde{\delta}.\tau_1 = \prod_i(\tau_1^i)\end{array}\right.$
3. $\forall\tilde{\alpha}.(\gamma,\sigma_1,\sigma_2) = \prod_i(\gamma_i,\sigma_1^i,\sigma_2^i) \Rightarrow \forall\tilde{\alpha}.(\gamma,\sigma_1) = \prod_i(\gamma_i,\sigma_1^i)$

Proof

proof is by induction on P.

case x: x **is a reference** We have, by assumption, that $\forall\tilde{\alpha}.(\gamma,\sigma) = (\gamma_1,\tau_1) \sqcap (\gamma_2,\tau_2)$ and $\Gamma_i(x) = \sigma_x^i \prec \sigma_i$. Or $\sigma_x^1 \sqcap \sigma_x^2 \prec \tau_i$ $(i \in \{1,2\})$ $\Rightarrow$ $\sigma_x^1 \sqcap \sigma_x^2 \prec \tau_1 \sqcap \tau_2$. Therefore, using Lemma B.1, proposition 1, with $\Gamma(x) =_{\text{def}} \sigma_x$, we prove that the following diagram holds

$$\begin{array}{ccc}\forall\tilde{\alpha}.\sigma_x & \xrightarrow{\sim} & \sigma_x^1 \sqcap \sigma_x^2\\ & & \downarrow \prec\\ \forall\tilde{\alpha}.\sigma & \xleftarrow[\sim]{} & \tau_1 \sqcap \tau_2\end{array}$$

Then $\forall\tilde{\alpha}.\sigma_x \prec \forall\tilde{\alpha}.\sigma$, which implies $\sigma_x \prec \sigma \prec \tau$, where $\sigma = \forall\tilde{\delta}.\tau$ and $\tilde{\delta} \cap \text{fn}(\gamma) = \emptyset$. Using the (taut) rule, we have $\Gamma(x) = \sigma_x \prec \tau \Rightarrow \Gamma \vdash_{\mathtt{M}} x : \tau$ as required.

x **is a variable** We use the previous notations with the differences that the σ_x^i's are monomorphic ($\sigma_x^i = \tau_i$) and that $\Gamma(x) = \tau_x$ with $\sigma_x = \forall\tilde{\alpha}\tilde{\eta}.\tau_x \sim \sigma_x^1 \sqcap \sigma_x^2$

$$\begin{array}{ccc}\forall\tilde{\alpha}.(\forall\tilde{\eta}.\tau_x) & \xrightarrow{\sim} & \\ & & \tau_1 \sqcap \tau_2\\ \forall\tilde{\alpha}.(\forall\tilde{\delta}.\tau) & \xleftarrow{\sim} & \end{array}$$

Then, it is sufficient to choose $\tilde{\delta} = \tilde{\eta}$ to prove the result.

case $(P\ u)$**:** We suppose here that u is a reference since, like in the previous case, we can easily adapt the proof if u is a variable. It must be the case that there is τ'_1 and τ'_2 such that $\Gamma_i \vdash_{\mathtt{M}} P : \tau'_i \to \tau_i$ with $\Gamma_i(u) = \sigma^i_u \prec \tau'_i$ $(i \in \{1,2\})$. Using the induction hypothesis, it follows that $\Gamma \vdash_{\mathtt{M}} P : \tau$ with $\forall\tilde{\alpha}.(\gamma, \forall\tilde{\delta}.\tau) = \prod_i(\gamma_i, \tau'_i \to \tau_i)$. Moreover, using Lemma B.1, proposition 2, there is ω' and ω such that $\tau = \omega' \to \omega$, $\forall\tilde{\alpha}.(\gamma, \forall\tilde{\delta}.\omega) = \prod(\gamma_i, \tau_i)$ and $\forall\tilde{\alpha}\tilde{\delta}.\omega' = \prod(\tau'_i)$.Therefore $\Gamma \vdash_{\mathtt{M}} P : \omega' \to \omega$ and, using the proof from the previous case, we have $\Gamma(u) = \sigma_u \prec \forall\tilde{\delta}.\omega' \prec \omega'$. The result follows after application of the (appl) rule $\Gamma \vdash_{\mathtt{M}} (P\ u) : \tau$.

case $(\lambda x)P$**:** There is τ'_i such that $\Gamma_i, x : \tau'_i \vdash_{\mathtt{M}} P : \tau_i$. Let $\forall\tilde{\alpha}.(\gamma, \sigma) = \prod(\gamma_i, \tau'_i \to \tau_i)$ and $\sigma = \forall\tilde{\delta}.\tau$ with $\tilde{\delta}$ fresh variables. Using Lemma B.1, proposition 2, it follows that there is ω' and ω such that $\tau = (\omega' \to \omega)$. Moreover, $\forall\tilde{\alpha}.(\gamma, \forall\tilde{\delta}.\omega', \forall\tilde{\delta}.\omega) = \prod_i(\gamma_i, \tau'_i, \tau_i)$. Using the induction hypothesis (x is a variable), we prove that $\Gamma, x : \tau' \vdash_{\mathtt{M}} P : \tau$. Remark that this is coherent with the proof for the first case since $\forall\tilde{\delta}.\omega'$ and $\forall\tilde{\delta}.\omega$ have the same quantified variables. The result follows using the (abs) rule $\Gamma \vdash_{\mathtt{M}} (\lambda x)P : \omega' \to \omega$ with $\forall\tilde{\alpha}.(\gamma, \forall\tilde{\delta}.(\omega' \to \omega)) = \prod_i(\gamma_i, \tau'_i \to \tau_i)$.

case $(P_1 \mid P_2)$**:** We have, by assumption, that $\Gamma_i \vdash_{\mathtt{M}} (P_1 \mid P_2) : \tau_i$. Therefore $\Gamma_i \vdash_{\mathtt{M}} P_1 : \tau_i$ (resp. P_2). Using induction we have that $\Gamma \vdash_{\mathtt{M}} P_1 : \tau$ (resp. P_2) with $\forall\tilde{\alpha}.(\gamma, \forall\tilde{\delta}.\tau) = \prod_i(\gamma_i, \tau_i)$. The result follows by using the (par) rule $\Gamma \vdash_{\mathtt{M}} (P_1 \mid P_2) : \tau$.

case $(\nu u)P$**:** It must be the case that there are two type schemes, ρ_1 and ρ_2, such that $\Gamma_i, x : \rho_i \vdash_{\mathtt{M}} P : \tau_i$. Let $\forall\tilde{\alpha}.(\gamma, \rho, \sigma) = \prod_i(\gamma_i, \rho_i, \tau_i)$ and $\sigma = \forall\tilde{\delta}.\tau$ with $\tilde{\delta}$ fresh. Using induction, we have that $\Gamma, u : \rho \vdash_{\mathtt{M}} P : \tau$ and using Lemma B.1, proposition 3, we have that $\forall\tilde{\alpha}.(\gamma, \forall\tilde{\delta}.\tau) = \prod_i(\gamma_i, \tau_i)$. Therefore the result follows by using the (new) rule $\Gamma \vdash_{\mathtt{M}} (\nu u)P : \tau$.

case $\langle u \Leftarrow P\rangle$**:** We have, by assumption, that $\Gamma_i \vdash_{\mathtt{M}} P : \tau_i$. It must be the case that $\Gamma_i \vdash_{\mathtt{M}} P : \omega_i$ with $\Gamma_i(u) = \sigma^i_u = \mathrm{Gen}_{\Gamma_i}(\omega_i)$. Let $\forall\tilde{\alpha}.(\gamma, \sigma) = \prod_i(\gamma_i, \omega_i)$ and $\Gamma(u) = \sigma_u$. Using Lemma B.1, proposition 1, we have that

$$\begin{array}{lcl} \forall\tilde{\alpha}.\sigma_u & \sim & \prod_i \sigma^i_u = \prod_i \mathrm{Gen}_{\Gamma_i}(\omega_i) \\ \forall\tilde{\alpha}.\sigma & \sim & \prod_i \omega_i \end{array}$$

We first show $\sigma_u \sim \mathrm{Gen}_\Gamma(\sigma)$ proving $\sigma_u \prec \mathrm{Gen}_\Gamma(\sigma)$ and $\mathrm{Gen}_\Gamma(\sigma) \prec \sigma_u$

1. $\sigma^i_u \prec \omega_i, i \in \{1,2\}$ $\Rightarrow$ $\forall\tilde{\alpha}.\sigma_u \prec \omega_1 \sqcap \omega_2$
   $\Rightarrow$ $\forall\tilde{\alpha}.\sigma_u \prec \forall\tilde{\alpha}.\sigma$
   $\Rightarrow$ $\sigma_u \prec \sigma$
   $\Rightarrow$ $\sigma_u \prec \mathrm{Gen}_\Gamma(\sigma)$

   The last step of this deduction is proved by reducing it to the absurd. Suppose $\forall\tilde{\alpha}.\sigma_u \not\prec \mathrm{Gen}_\Gamma(\sigma)$. Since $\sigma_u \prec \sigma$, there must be a generic type variable, ϵ, free in both σ and σ_u and not in $\mathrm{fn}(\Gamma)$. Or $\epsilon \in \mathrm{fn}(\sigma)$ (resp. $\epsilon \in \mathrm{fn}(\sigma_u)$) implies ϵ free

in ω_i ($i \in \{1,2\}$) (resp. $\mathrm{Gen}_{\Gamma_i}(\omega_i)$) and $\epsilon \notin \mathrm{fn}(\Gamma)$ implies $\epsilon \notin (\mathrm{fn}(\Gamma_1) \cap \mathrm{fn}(\Gamma_2))$. Then ϵ is generalized in one of the $\mathrm{Gen}_{\Gamma_i}(\omega_i)$, which contradicts the fact that ϵ is free in $\mathrm{Gen}_{\Gamma_1}(\omega_1)$ and $\mathrm{Gen}_{\Gamma_2}(\omega_2)$.

2. $\forall \tilde{\alpha}.\sigma \prec \omega_i, i \in \{1,2\} \quad \Rightarrow \quad \mathrm{Gen}_\Gamma(\forall \tilde{\alpha}.\sigma) \prec \mathrm{Gen}_{\Gamma_i}(\omega_i), i \in \{1,2\}$
$\Rightarrow \quad \mathrm{Gen}_\Gamma(\forall \tilde{\alpha}.\sigma) \prec \prod_i \mathrm{Gen}_{\Gamma_i}(\omega_i)$
$\Rightarrow \quad \mathrm{Gen}_\Gamma(\forall \tilde{\alpha}.\sigma) \prec \forall \tilde{\alpha}.\sigma_u$
$\Rightarrow \quad \sigma \prec \sigma_u$

Here the first step deserves a proof. This is left to the reader since it is basically the same argument that in the previous demonstration.

Let ω be the instantiation of σ to fresh variables. Using induction we have $\Gamma \vdash_{\mathtt{M}} P : \omega$ with $\Gamma(u) \sim \mathrm{Gen}_\Gamma(\omega)$. By applying rule (decl+), it follows that for every type τ, $\Gamma \vdash_{\mathtt{M}} \langle u \Leftarrow P \rangle : \tau$. For example, let τ be a type such that $\forall \tilde{\alpha}.(\gamma', \forall \tilde{\eta}.\tau) = \prod_i(\gamma_i, \tau_i)$. Using Lemma B.1, proposition 3, we have $\gamma' = \gamma$. Therefore $\Gamma \vdash_{\mathtt{M}} P : \tau$ with $\forall \tilde{\alpha}.(\gamma, \forall \tilde{\eta}.\tau) = \prod_i(\gamma_i, \tau_i)$.

□

References

[1] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press.

[2] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[3] M. Boreale. On the expresiveness of internal mobility in name-passing calculi. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[4] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, I.N.R.I.A., 1992.

[5] G. Boudol. The lambda-calculus with multiplicities. Technical Report 2025, I.N.R.I.A., September 1993.

[6] G. Boudol and C. Laneve. the discriminating power of multiplicities in the λ-calculus. Technical Report 2441, I.N.R.I.A., December 1994.

[7] G. Boudol and C. Laneve. λ-calculus, multiplicities and the π-calculus. Technical Report 2581, I.N.R.I.A., 1995.

[8] Gérard Boudol. The π-calculus in direct style. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, Paris, France, 15–17 January 1997.

[9] L. Cardelli and J. C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991. Summary in Math. Foundations of Prog. Lang. Semantics, Springer LNCS 442, 1990, pp 22–52.

[10] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal description of Programming Concepts*, pages 431–507. Springer-Verlag, 1989.

[11] Felice Cardone and Mario Coppo. Two extension of Curry's type inference system. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[12] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the ACM Conference on Lisp and Functionnal Programming*. ACM, 1986.

[13] Luis Damas and Robin Milner. Principal type schemes for functional programming. In *9th Symposium on Principles of Programming Languages*, 1982.

[14] Luis Manuel Martins Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, 1984.

[15] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag.

[16] Cédric Fournet, Luc Maranget, Cosimo Laneve, and Didier Rémy. Implicit typing à la ML for the join-calculus. In *8th International Conference on Concurrency Theory (CONCUR'97)*, Warsaw, Poland, 1997. to appear.

[17] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6:361–380, 1993.

[18] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(254–289), April 1993.

[19] K. Honda. Two bisimilarities for the ν-calculus. Technical Report 92-002, Department of Computer Science, Keio University, 1992.

[20] Trevor Jim. What are principal typings and what are they good for? In *Proceedings of POPL'96*, 1996.

[21] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proceedings of the 22nd annual ACM symposium on theory of Computation (STOC)*, pages 468–476, New-York, 1990. ACM.

[22] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[23] Carolina Lavatelli. *Sémantique du lambda calcul avec ressources.* PhD thesis, Université Paris VII, January 1996.

[24] Xavier Leroy. Polymorphic typing of an algorithmic language. Technical report, I.N.R.I.A., INRIA-Rocquencourt, Le Chesnay, France, 1992. Ph.D. thesis.

[25] Xavier Leroy. Polymorphism by name for references and continuations. In *Proc. 20th symp. Principles of Programming Languages*, pages 220–231. ACM Press, 1993.

[26] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *17th Symposium on Principles of Programming Languages.* ACM Press, January 1990.

[27] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[28] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[29] Alan Mycroft. Polymorphic type scheme and recursive definitions. In *6th International Symposium on programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[30] Alan Mycroft. Incremental polymorphic type checking with update. In *Proc. LFCS'92, 2nd International Symposium Tver'92*, volume 620 of *Lecture Notes in Computer Science.* Springer-Verlag, 1992.

[31] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[32] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 11, 1995.

[33] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, I.N.R.I.A., May 1991.

[34] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[35] David N. Turner. *The polymorphic pi-calculus: theory and implementation.* PhD thesis, University of Edinburgh, 1995.

[36] Pawel Urzyczyn. Polymorphic type checking and related problems, 1996. (lectures notes).

[37] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic pi-calculus. In *4th International conference on concurrency theory*, volume 715, pages 524–538. Springer-Verlag, 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399